# Problem 1

Consider a 2D robotic arm with 3 links. The position of its end-effector is governed by the arm lengths and joint angles as follows (as in the figure "data/robot-arm.png"): $$ x = L_1 \cos(\theta_1) + L_2\cos(\theta_2+\theta_1) + L_3\cos(\theta_3+\theta_2+\theta_1)\\ y = L_1\sin(\theta_1) + L_2\sin(\theta_2+\theta_1) + L_3\sin(\theta_3+\theta_2+\theta_1) $$

In robotics settings, inverse-kinematics problems are common for setups like this. For example, suppose all 3 arm lengths are $L_1 = L_2 = L_3 = 1$, and we want to position the end-effector at $(x,y) = (0.5, 0.5)$. What set of joint angles $(\theta_1, \theta_2, \theta_3)$ should we choose for the end-effector to reach this position?

In this problem you will train a neural network to find a function mapping from coordinates $(x,y)$ to joint angles $(\theta_1, \theta_2, \theta_3)$ that position the end-effector at $(x,y)$.

### Summary of deliverables:

1. Neural network model
2. Generate training and validation data
3. Training function
4. 6 plots with training and validation loss
5. 6 prediction plots
6. Respond to the prompts

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt

import torch
from torch import nn, optim

class ForwardArm(nn.Module):
    def __init__(self, L1=1, L2=1, L3=1):
        super().__init__()
```

```python
        self.L1 = L1
        self.L2 = L2
        self.L3 = L3
    def forward(self, angles):
        theta1 = angles[:,0]
        theta2 = angles[:,1]
        theta3 = angles[:,2]
        x = self.L1*torch.cos(theta1) + self.L2*torch.cos(theta1+theta2) + self.L3*torch.cos(theta1+theta2+theta3)
        y = self.L1*torch.sin(theta1) + self.L2*torch.sin(theta1+theta2) + self.L3*torch.sin(theta1+theta2+theta3)
        return torch.vstack([x,y]).T

def plot_predictions(model, title=""):
    fwd = ForwardArm()

    vals = np.arange(0.1, 2.0, 0.2)
    x, y = np.meshgrid(vals,vals)
    coords = torch.tensor(np.vstack([x.flatten(),y.flatten()]).T,dtype=torch.float)
    angles = model(coords)
    preds = fwd(angles).detach().numpy()

    plt.figure(figsize=[4,4],dpi=140)

    plt.scatter(x.flatten(), y.flatten(), s=60, c="None",marker="o",edgecolors="k", label="Targets")
    plt.scatter(preds[:,0], preds[:,1], s=25, c="red", marker="o", label="Predictions")
    plt.text(0.1, 2.15, f"MSE = {nn.MSELoss()(fwd(model(coords)),coords):.1e}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim(-.1,2.1)
    plt.ylim(-.1,2.4)
    plt.legend()
    plt.title(title)
    plt.show()

def plot_arm(theta1, theta2, theta3, L1=1,L2=1,L3=1, show=True):
    x1 = L1*np.cos(theta1)
    y1 = L1*np.sin(theta1)
    x2 = x1 + L2*np.cos(theta1+theta2)
    y2 = y1 + L2*np.sin(theta1+theta2)
    x3 = x2 + L3*np.cos(theta1+theta2+theta3)
    y3 = y2 + L3*np.sin(theta1+theta2+theta3)
    xs = np.array([0,x1,x2,x3])
    ys = np.array([0,y1,y2,y3])

    plt.figure(figsize=(5,5),dpi=140)
    plt.plot(xs, ys, linewidth=3, markersize=5,color="gray", markerfacecolor="lightgray",marker="o",markeredgecolor="bl
```

```
    plt.scatter(x3,y3,s=50,color="blue",marker="P",zorder=100)
    plt.scatter(0,0,s=50,color="black",marker="s",zorder=-100)

    plt.xlim(-1.5,3.5)
    plt.ylim(-1.5,3.5)

    if show:
        plt.show()
```

# End-effector position

You can use the interactive figure below to visualize the robot arm.

In [2]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, FloatSlider, Dropdown

def plot_unit_arm(theta1, theta2, theta3):
    plot_arm(theta1, theta2, theta3)

slider1 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta1',disabled=False,con
slider2 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta2',disabled=False,con
slider3 = FloatSlider(value=0, min=-np.pi*0.75, max=np.pi*0.75, step=np.pi/100, description='theta3',disabled=False,con

interactive_plot = interactive(plot_unit_arm, theta1 = slider1, theta2 = slider2, theta3 = slider3)
output = interactive_plot.children[-1]
output.layout.height = '600px'

interactive_plot
```

Out[2]:
```
interactive(children=(FloatSlider(value=0.0, description='theta1', layout=Layout(width='550px'), max=2.3561944…
```

# Neural Network for Inverse Kinematics

In this class we have mainly had regression problems with only one output. However, you can create neural networks with any number of outputs just by changing the size of the last layer. For this problem, we already know the function to go from joint angles (3) to end-effector coordinates (2). This is provided in neural network format as `ForwardArm()`.

If you provide an instance of `ForwardArm()` with an $N\times3$ tensor of joint angles, and it will return an $N\times2$ tensor of coordinates.

Here, you should create a neural network with 2 inputs and 3 outputs that, once trained, can output the joint angles (in radians) necessary to reach the input x-y coordinates.

In the cell below, complete the definition for `InverseArm()`:

- The initialization argument `hidden_layer_sizes` dictates the number of neurons per hidden layer in the network. For example, `hidden_layer_sizes=[12,24]` should create a network with 2 inputs, 12 neurons in the first hidden layer, 24 neurons in the second hidden layer, and 3 outputs.
- Use a ReLU activation at the end of each hidden layer.
- The initialization argument `max_angle` refers to the maximum bend angle of the joint. If `max_angle=None`, there should be no activation at the last layer. However, if `max_angle=1` (for example), then the output joint angles should be restricted to the interval [-1, 1] (radians). You can clamp values with the tanh function (and then scale them) to achieve this.

In [3]:
```
class InverseArm(nn.Module):
    def __init__(self, hidden_layer_sizes=[24,24], max_angle=None):
        super().__init__()
        # YOUR CODE GOES HERE
        N_in = 2
        N_out = 3
        size = len(hidden_layer_sizes)
        layer = []
        layer.append(nn.Linear(N_in,hidden_layer_sizes[0]))
        layer.append(nn.ReLU())
        for i in range(size):
            if i>0:
                layer.append(nn.Linear(hidden_layer_sizes[i-1],hidden_layer_sizes[i]))
                layer.append(nn.ReLU())
        layer.append(nn.Linear(hidden_layer_sizes[size-1],N_out))
        self.seq = nn.Sequential(*layer)
        self.max_angle = max_angle

    def forward(self, xy):
        # YOUR CODE GOES HERE
        xy = self.seq(xy)
        if self.max_angle is not None:
            xy = nn.Tanh()(xy)
            xy = self.max_angle* xy
        return xy
```

# Generate Data

In the cell below, generate a dataset of x-y coordinates. You should use a $100\times 100$ meshgrid, for x and y each on the interval $[0, 2]$.

Randomly split your data so that 80% of points are in `X_train`, while the remaining 20% are in `X_val`. (Each of these should have 2 columns -- x and y)

In [4]:
```
# YOUR CODE GOES HERE
from sklearn.model_selection import train_test_split

x = np.linspace(0,2,100)
y = np.linspace(0,2,100)
X,Y = np.meshgrid(x,y)
data = np.column_stack((X.ravel(),Y.ravel()))
X_train,X_val = train_test_split(data, test_size = 0.2)
X_train = torch.Tensor(X_train)
X_val = torch.Tensor(X_val)
print(X_train.shape)
print(X_val.shape)
```

```
torch.Size([8000, 2])
torch.Size([2000, 2])
```

# Training function

Write a function `train()` below with the following specifications:

*Inputs:*

- `model` : `InverseArm` model to train
- `X_train` : $N\times 2$ vector of training x-y coordinates
- `X_val` : $N\times 2$ vector of validation x-y coordinates
- `lr` : Learning rate for Adam optimizer
- `epochs` : Total epoch count
- `gamma` : ExponentialLR decay rate
- `create_plot` : ( `True` / `False` ) Whether to display a plot with training and validation loss curves

*Loss function:*
The loss function you use should be based on whether the end-effector moves to the correct location. It should be the MSE between

the target coordinate tensor and the coordinates that the predicted joint angles produce. In other words, if your inverse kinematics model is `model`, and `fwd` is an instance of `ForwardArm()`, then you want the MSE between input coordinates `X` and `fwd(model(X))`.

In [5]:
```
from torch.optim.lr_scheduler import ExponentialLR

def train(model, X_train, X_val, lr = 0.01, epochs = 1000, gamma = 1, create_plot = True):
    # YOUR CODE GOES HERE
    loss_fcn = nn.MSELoss()
    opt = optim.Adam(params = model.parameters(), lr=lr)
    scheduler = ExponentialLR(optimizer=opt, gamma=gamma)
    fwd = ForwardArm()
    train_hist = []
    val_hist = []

    for epoch in range(epochs+1):
        model.train()
        out_train = model(X_train)
        y_train = fwd(out_train)
        #print(y)

        loss_train = loss_fcn(y_train, X_train)
        train_hist.append(loss_train.item())

        model.eval()
        #print(model(fwd(model(X_val))))
        out_val = model(X_val)
        y_val = fwd(out_val)
        loss_val = loss_fcn(y_val, X_val)
        val_hist.append(loss_val.item())

        opt.zero_grad()
        loss_train.backward()
        opt.step()
        scheduler.step()
        if epoch % int(epochs / 25) == 0:
            print(f"Epoch {epoch:>4} of {epochs}:   Train Loss = {loss_train.item():.4f}   Validation Loss = {loss_val.

    if create_plot is True:
        plt.figure(figsize=(4,2),dpi=250)
        plt.plot(train_hist,label="Training")
        plt.plot(val_hist,label="Validation",linewidth=1)
        plt.legend()
        plt.xlabel("Epoch")
```

```
        plt.ylabel("Loss")
        plt.show()
    return model
```

# Training a model

Create 3 models of different complexities (with `max_angle=None`):

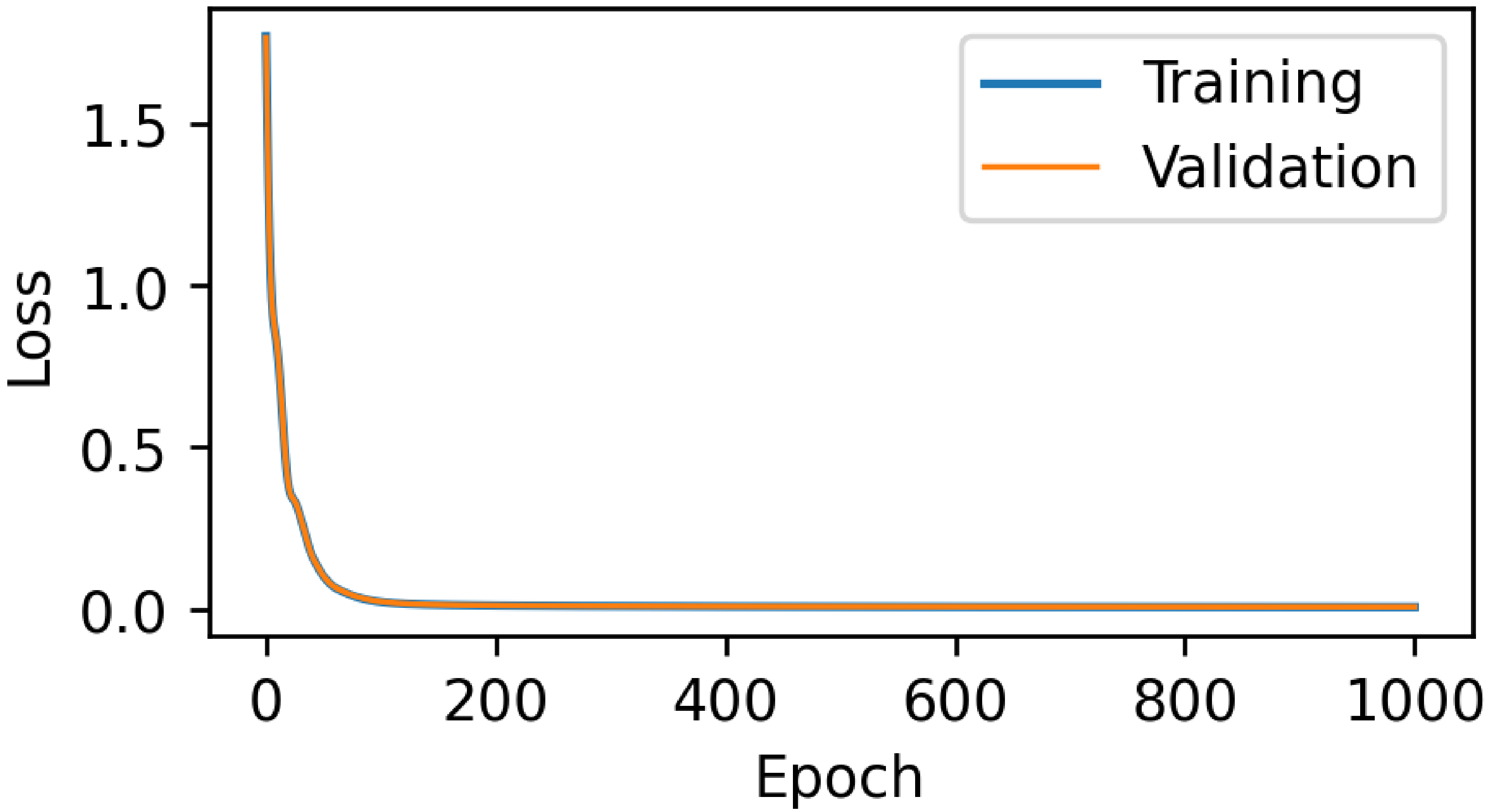
- `hidden_layer_sizes=[12]`
- `hidden_layer_sizes=[24,24]`
- `hidden_layer_sizes=[48,48,48]`

Train each model for 1000 epochs, learning rate 0.01, and gamma 0.995. Show the plot for each.

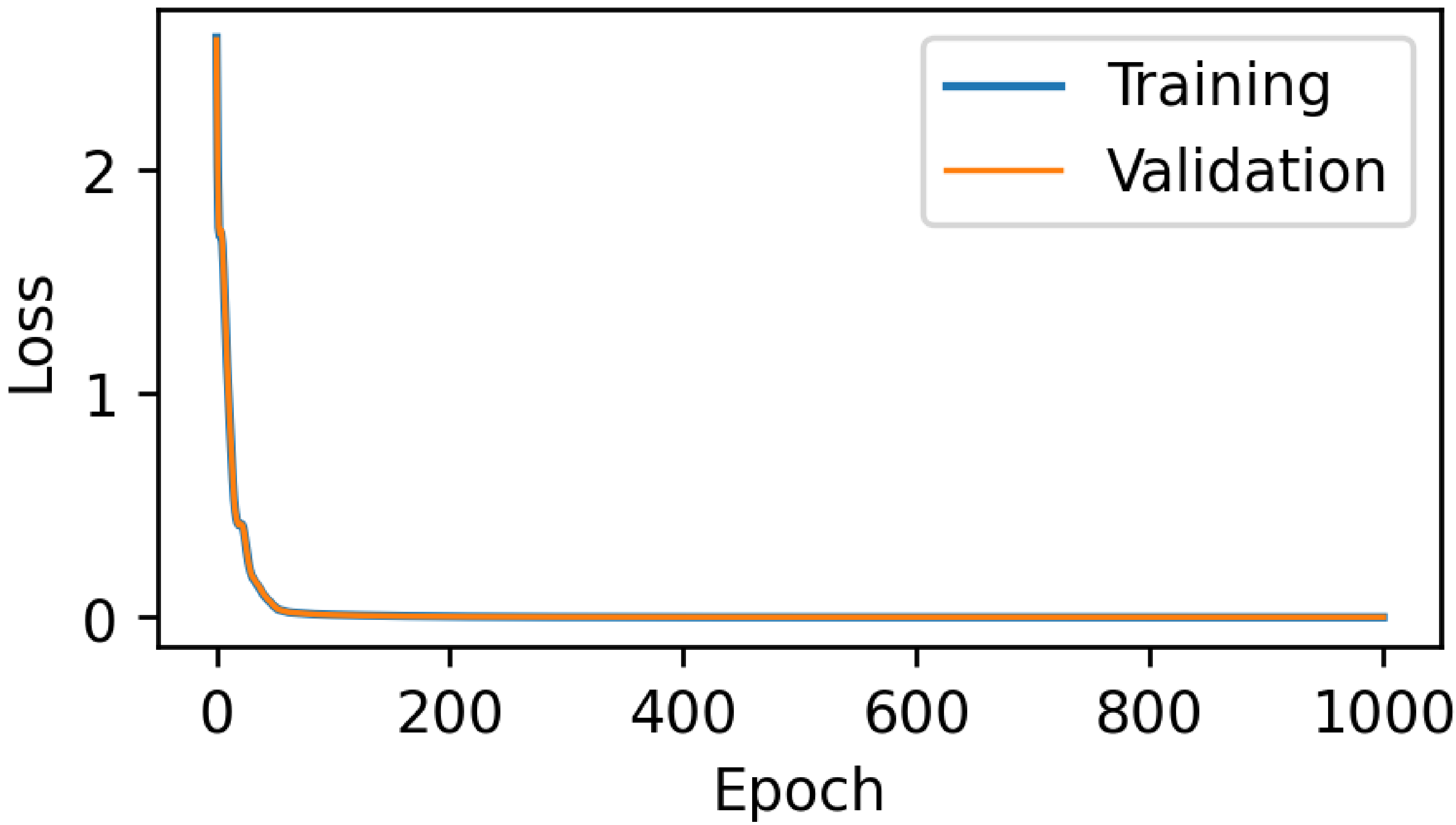
In [6]:

```
# YOUR CODE GOES HERE
model1 = train(model = InverseArm(hidden_layer_sizes = [12]),X_train = X_train,X_val = X_val,
               lr = 0.01,epochs = 1000, gamma = 0.995,create_plot = True)
model2 = train(model = InverseArm(hidden_layer_sizes = [24,24]),X_train = X_train,X_val = X_val,
               lr = 0.01,epochs = 1000, gamma = 0.995,create_plot = True)
model3 = train(model = InverseArm(hidden_layer_sizes = [48,48,48]),X_train = X_train,X_val = X_val,
               lr = 0.01,epochs = 1000, gamma = 0.995,create_plot = True)
```

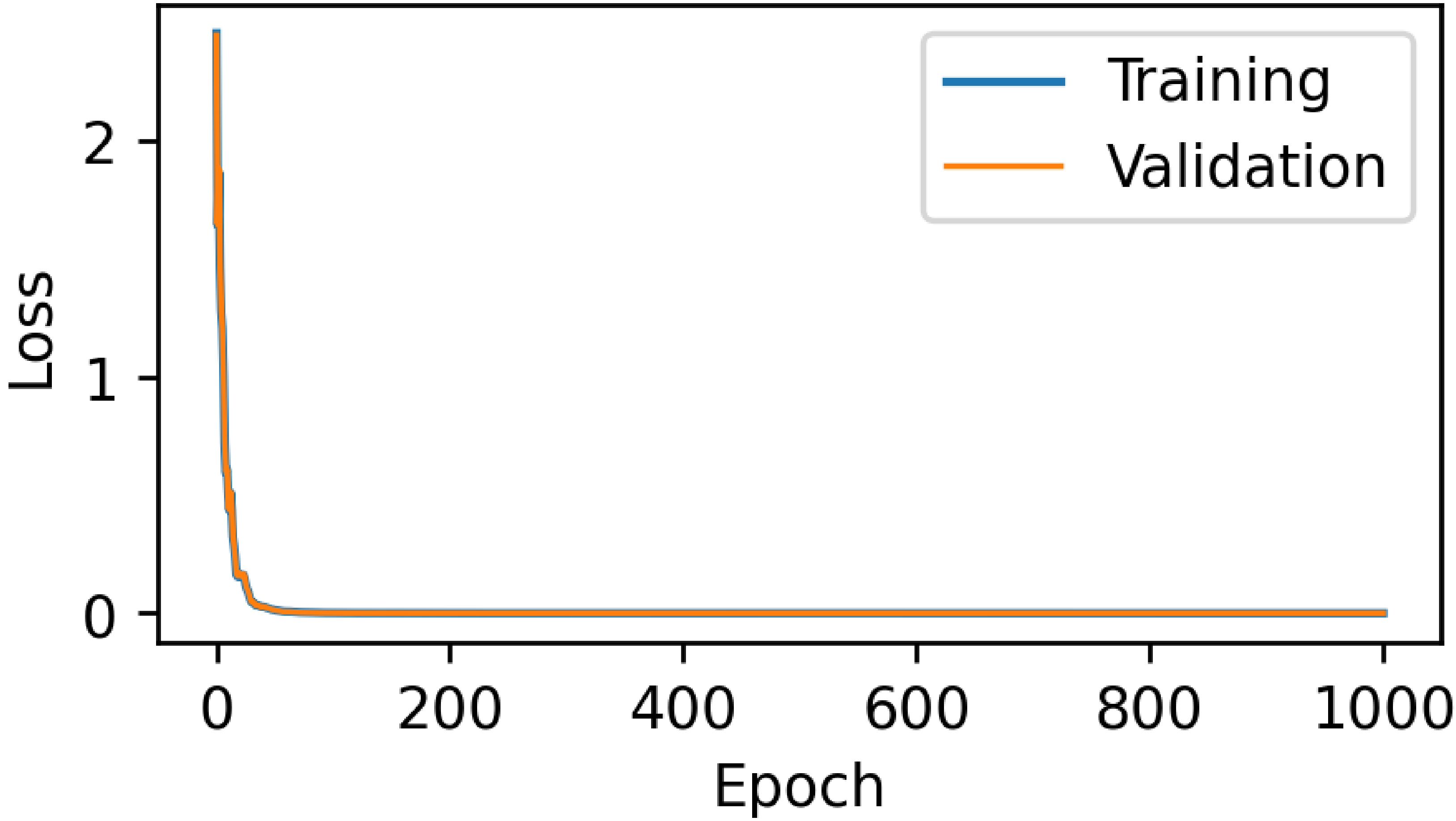
```
Epoch    0 of 1000:   Train Loss = 1.7717   Validation Loss = 1.7668
Epoch   40 of 1000:   Train Loss = 0.1650   Validation Loss = 0.1651
Epoch   80 of 1000:   Train Loss = 0.0359   Validation Loss = 0.0371
Epoch  120 of 1000:   Train Loss = 0.0161   Validation Loss = 0.0168
Epoch  160 of 1000:   Train Loss = 0.0122   Validation Loss = 0.0127
Epoch  200 of 1000:   Train Loss = 0.0107   Validation Loss = 0.0111
Epoch  240 of 1000:   Train Loss = 0.0098   Validation Loss = 0.0102
Epoch  280 of 1000:   Train Loss = 0.0092   Validation Loss = 0.0095
Epoch  320 of 1000:   Train Loss = 0.0087   Validation Loss = 0.0089
Epoch  360 of 1000:   Train Loss = 0.0083   Validation Loss = 0.0085
Epoch  400 of 1000:   Train Loss = 0.0080   Validation Loss = 0.0082
Epoch  440 of 1000:   Train Loss = 0.0077   Validation Loss = 0.0079
Epoch  480 of 1000:   Train Loss = 0.0075   Validation Loss = 0.0077
Epoch  520 of 1000:   Train Loss = 0.0073   Validation Loss = 0.0075
Epoch  560 of 1000:   Train Loss = 0.0072   Validation Loss = 0.0073
Epoch  600 of 1000:   Train Loss = 0.0070   Validation Loss = 0.0072
Epoch  640 of 1000:   Train Loss = 0.0069   Validation Loss = 0.0071
Epoch  680 of 1000:   Train Loss = 0.0068   Validation Loss = 0.0070
Epoch  720 of 1000:   Train Loss = 0.0068   Validation Loss = 0.0069
Epoch  760 of 1000:   Train Loss = 0.0067   Validation Loss = 0.0069
Epoch  800 of 1000:   Train Loss = 0.0067   Validation Loss = 0.0068
Epoch  840 of 1000:   Train Loss = 0.0066   Validation Loss = 0.0068
Epoch  880 of 1000:   Train Loss = 0.0066   Validation Loss = 0.0067
Epoch  920 of 1000:   Train Loss = 0.0065   Validation Loss = 0.0067
Epoch  960 of 1000:   Train Loss = 0.0065   Validation Loss = 0.0067
Epoch 1000 of 1000:   Train Loss = 0.0065   Validation Loss = 0.0067
```


Training
Validation
Loss
1.5
1.0
0.5
0.0
0
200
400
600
800
1000
Epoch

```
Epoch    0 of 1000:   Train Loss = 2.5900   Validation Loss = 2.5817
Epoch   40 of 1000:   Train Loss = 0.1039   Validation Loss = 0.1071
Epoch   80 of 1000:   Train Loss = 0.0159   Validation Loss = 0.0155
Epoch  120 of 1000:   Train Loss = 0.0093   Validation Loss = 0.0090
Epoch  160 of 1000:   Train Loss = 0.0061   Validation Loss = 0.0059
Epoch  200 of 1000:   Train Loss = 0.0040   Validation Loss = 0.0039
Epoch  240 of 1000:   Train Loss = 0.0028   Validation Loss = 0.0028
Epoch  280 of 1000:   Train Loss = 0.0021   Validation Loss = 0.0021
Epoch  320 of 1000:   Train Loss = 0.0017   Validation Loss = 0.0017
Epoch  360 of 1000:   Train Loss = 0.0014   Validation Loss = 0.0014
Epoch  400 of 1000:   Train Loss = 0.0013   Validation Loss = 0.0012
Epoch  440 of 1000:   Train Loss = 0.0011   Validation Loss = 0.0011
Epoch  480 of 1000:   Train Loss = 0.0011   Validation Loss = 0.0010
Epoch  520 of 1000:   Train Loss = 0.0010   Validation Loss = 0.0010
Epoch  560 of 1000:   Train Loss = 0.0010   Validation Loss = 0.0009
Epoch  600 of 1000:   Train Loss = 0.0009   Validation Loss = 0.0009
Epoch  640 of 1000:   Train Loss = 0.0009   Validation Loss = 0.0009
Epoch  680 of 1000:   Train Loss = 0.0009   Validation Loss = 0.0009
Epoch  720 of 1000:   Train Loss = 0.0009   Validation Loss = 0.0008
Epoch  760 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
Epoch  800 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
Epoch  840 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
Epoch  880 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
Epoch  920 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
Epoch  960 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
Epoch 1000 of 1000:   Train Loss = 0.0008   Validation Loss = 0.0008
```


Training
Validation
Loss
2
1
0
0
200
400
600
800
1000
Epoch

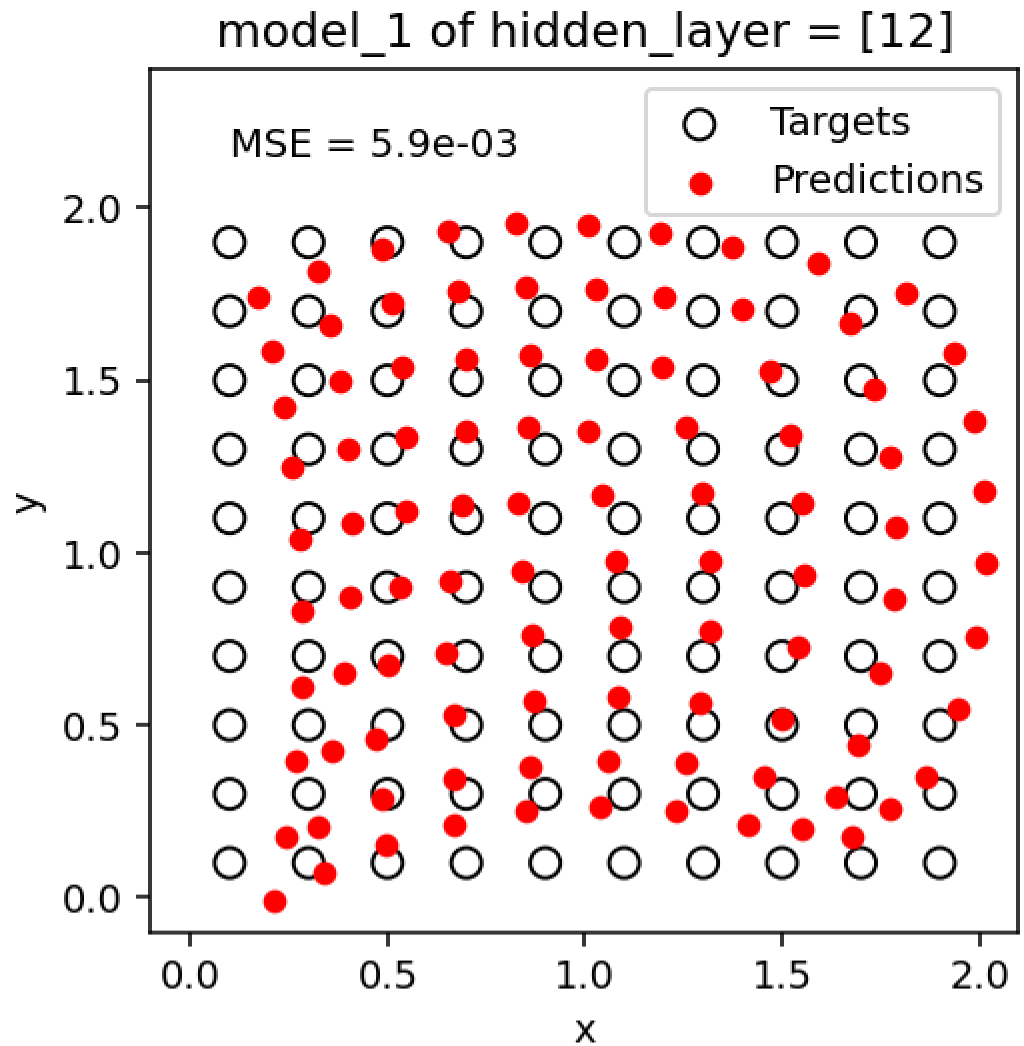
```
Epoch    0 of 1000:   Train Loss = 2.4561   Validation Loss = 2.4485
Epoch   40 of 1000:   Train Loss = 0.0281   Validation Loss = 0.0284
Epoch   80 of 1000:   Train Loss = 0.0027   Validation Loss = 0.0026
Epoch  120 of 1000:   Train Loss = 0.0010   Validation Loss = 0.0010
Epoch  160 of 1000:   Train Loss = 0.0007   Validation Loss = 0.0007
Epoch  200 of 1000:   Train Loss = 0.0006   Validation Loss = 0.0006
Epoch  240 of 1000:   Train Loss = 0.0005   Validation Loss = 0.0005
Epoch  280 of 1000:   Train Loss = 0.0004   Validation Loss = 0.0004
Epoch  320 of 1000:   Train Loss = 0.0004   Validation Loss = 0.0004
Epoch  360 of 1000:   Train Loss = 0.0004   Validation Loss = 0.0004
Epoch  400 of 1000:   Train Loss = 0.0004   Validation Loss = 0.0004
Epoch  440 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  480 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  520 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  560 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  600 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  640 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  680 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  720 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  760 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  800 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  840 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  880 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  920 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch  960 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
Epoch 1000 of 1000:   Train Loss = 0.0003   Validation Loss = 0.0003
```

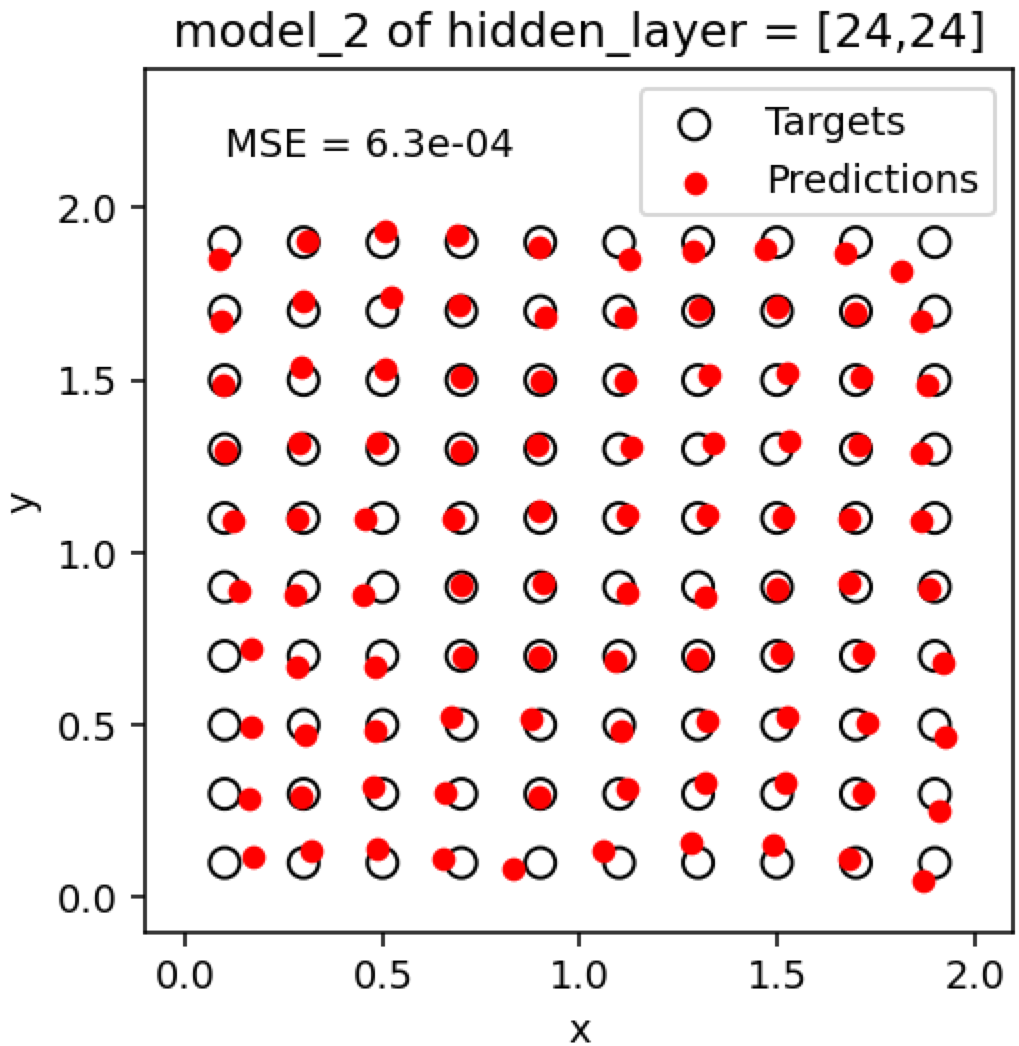


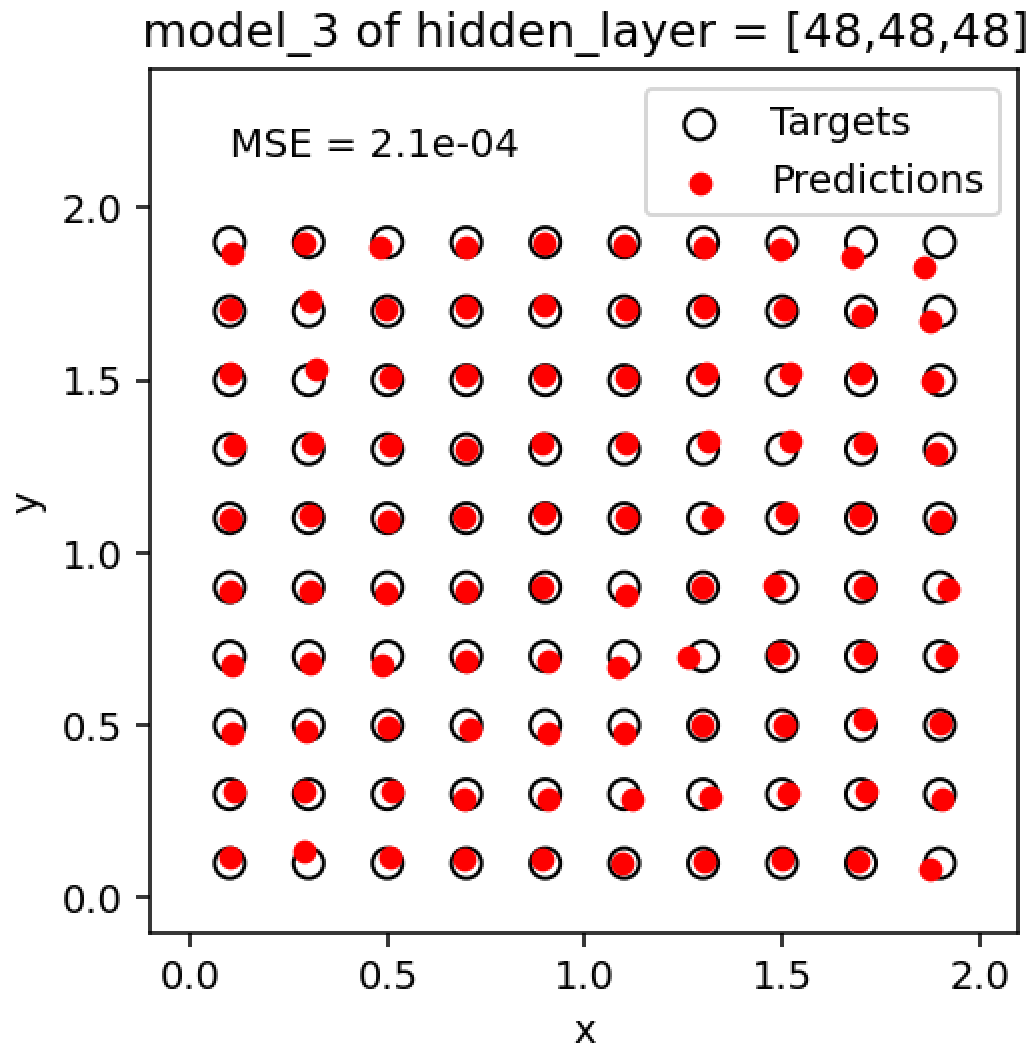

## Visualizations

For each of your models, use the function `plot_predictions` to visualize model predictions on the domain. You should observe improvements with increasing network size.

In [7]:
```
# YOUR CODE GOES HERE
plot_predictions(model1,title = "model_1 of hidden_layer = [12]")
plot_predictions(model2,title = "model_2 of hidden_layer = [24,24]")
plot_predictions(model3,title = "model_3 of hidden_layer = [48,48,48]")
```


model_1 of hidden_layer = [12]
MSE = 5.9e-03
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x


model_2 of hidden_layer = [24,24]
MSE = 6.3e-04
Targets
Predictions
0.0
0.5
1.0
1.5
2.0
x
y




## Interactive Visualization

You can use the interactive plot below to look at the performance of your model. (The model used must be named `model`.)

In [8]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, FloatSlider, Dropdown

def plot_inverse(x, y):
```

```
    xy = torch.Tensor([[x,y]])
    theta1, theta2, theta3 = model3(xy).detach().numpy().flatten().tolist()
    plot_arm(theta1, theta2, theta3, show=False)
    plt.scatter(x, y, s=100, c="red",zorder=1000,marker="x")
    plt.plot([0,2,2,0,0],[0,0,2,2,0],c="lightgray",linewidth=1,zorder=-1000)
    plt.show()

slider1 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100, description='x', disabled=False, continuous_update=True, o
slider2 = FloatSlider(value=1, min=-.5, max=2.5, step=1/100, description='y', disabled=False, continuous_update=True, o

interactive_plot = interactive(plot_inverse, x = slider1, y = slider2)
output = interactive_plot.children[-1]
output.layout.height = '600px'

interactive_plot
```

Out[8]:

```
interactive(children=(FloatSlider(value=1.0, description='x', layout=Layout(width='550px'), max=2.5, min=-0.5,…
```

# Training more neural networks

Now train more networks with the following details:

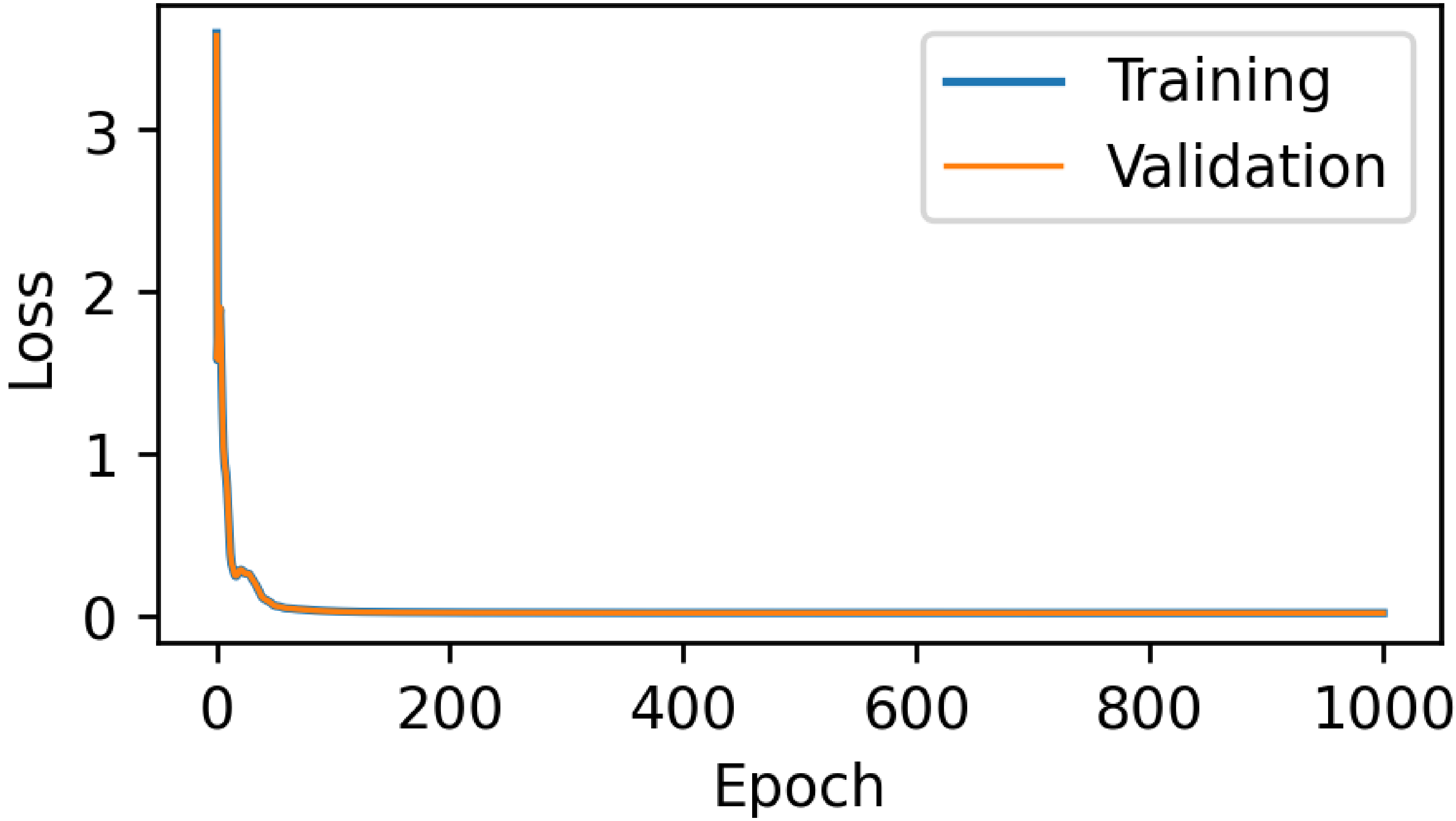
1. `hidden_layer_sizes=[48,48], max_angle=torch.pi/2`, train with `lr=0.01, epochs=1000, gamma=.995`
2. `hidden_layer_sizes=[48,48], max_angle=None`, train with `lr=1, epochs=1000, gamma=1`
3. `hidden_layer_sizes=[48,48], max_angle=2`, train with `lr=0.0001, epochs=300, gamma=1`

For each network, show a loss curve plot and a `plot_predictions` plot.

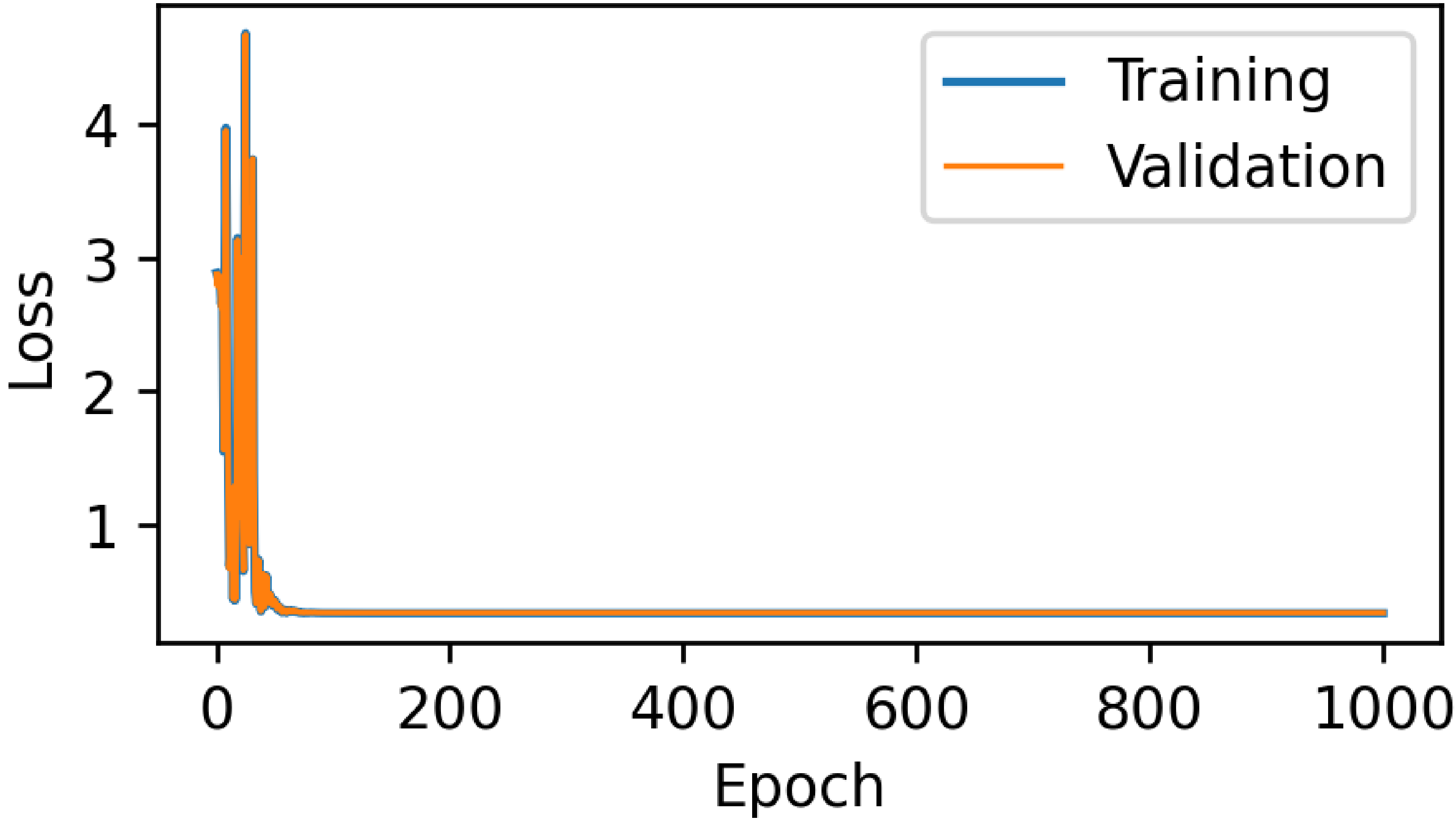
In [9]:

```
# YOUR CODE GOES HERE
model4 = train(model = InverseArm(hidden_layer_sizes = [48,48],max_angle = torch.pi/2),X_train = X_train,X_val = X_val,
               lr = 0.01,epochs = 1000, gamma = 0.995,create_plot = True)
model5 = train(model = InverseArm(hidden_layer_sizes = [48,48],max_angle = None),X_train = X_train,X_val = X_val,
               lr = 1,epochs = 1000, gamma = 1,create_plot = True)
model6 = train(model = InverseArm(hidden_layer_sizes = [48,48], max_angle = 2),X_train = X_train,X_val = X_val,
               lr = 0.0001,epochs = 300, gamma = 1,create_plot = True)
```

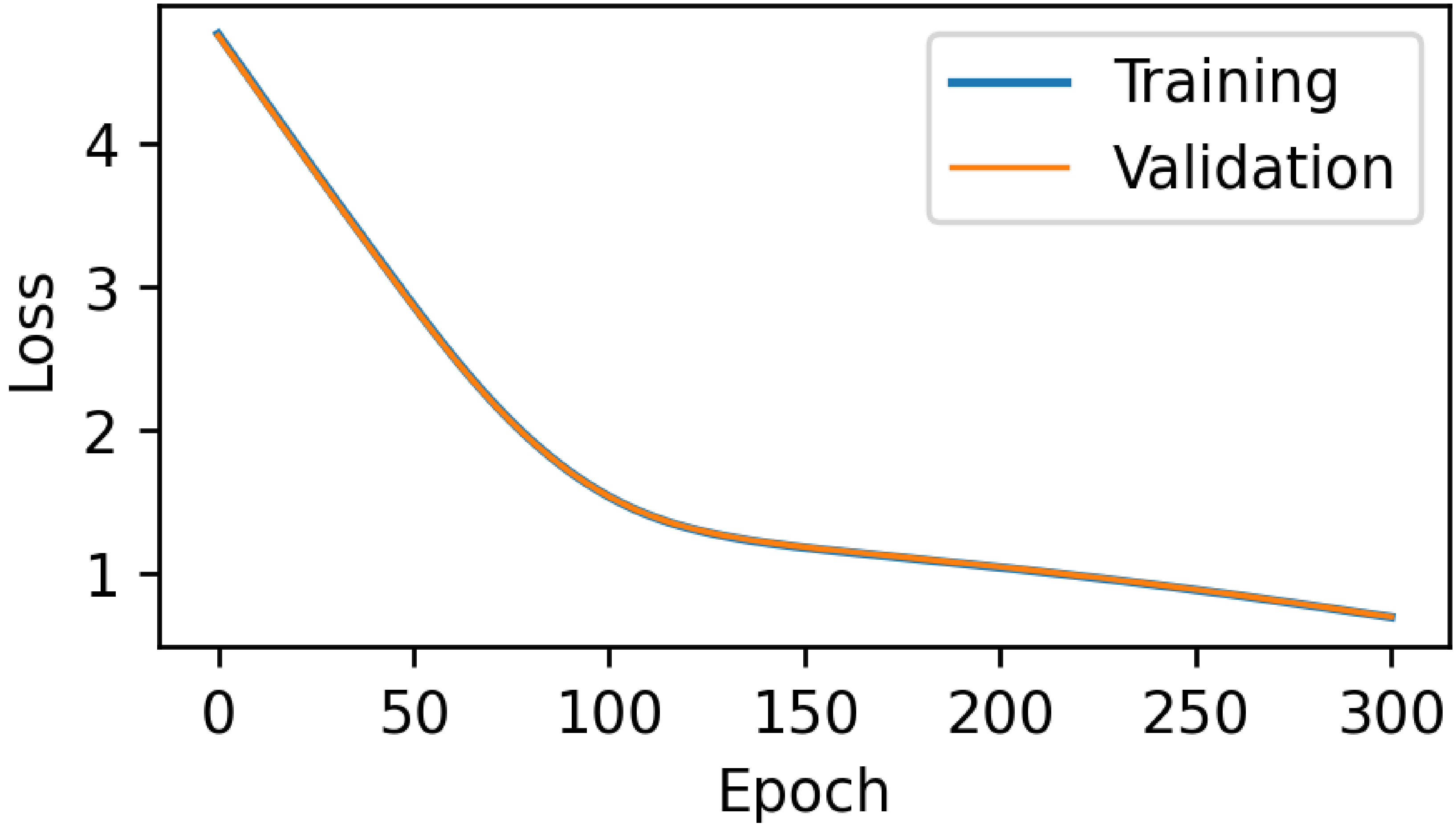
```
Epoch    0 of 1000:   Train Loss = 3.5927   Validation Loss = 3.5798
Epoch   40 of 1000:   Train Loss = 0.1106   Validation Loss = 0.1108
Epoch   80 of 1000:   Train Loss = 0.0376   Validation Loss = 0.0372
Epoch  120 of 1000:   Train Loss = 0.0261   Validation Loss = 0.0255
Epoch  160 of 1000:   Train Loss = 0.0233   Validation Loss = 0.0226
Epoch  200 of 1000:   Train Loss = 0.0221   Validation Loss = 0.0213
Epoch  240 of 1000:   Train Loss = 0.0213   Validation Loss = 0.0205
Epoch  280 of 1000:   Train Loss = 0.0208   Validation Loss = 0.0200
Epoch  320 of 1000:   Train Loss = 0.0205   Validation Loss = 0.0196
Epoch  360 of 1000:   Train Loss = 0.0202   Validation Loss = 0.0193
Epoch  400 of 1000:   Train Loss = 0.0200   Validation Loss = 0.0191
Epoch  440 of 1000:   Train Loss = 0.0198   Validation Loss = 0.0190
Epoch  480 of 1000:   Train Loss = 0.0197   Validation Loss = 0.0188
Epoch  520 of 1000:   Train Loss = 0.0196   Validation Loss = 0.0187
Epoch  560 of 1000:   Train Loss = 0.0196   Validation Loss = 0.0187
Epoch  600 of 1000:   Train Loss = 0.0195   Validation Loss = 0.0186
Epoch  640 of 1000:   Train Loss = 0.0195   Validation Loss = 0.0185
Epoch  680 of 1000:   Train Loss = 0.0194   Validation Loss = 0.0185
Epoch  720 of 1000:   Train Loss = 0.0194   Validation Loss = 0.0185
Epoch  760 of 1000:   Train Loss = 0.0194   Validation Loss = 0.0184
Epoch  800 of 1000:   Train Loss = 0.0193   Validation Loss = 0.0184
Epoch  840 of 1000:   Train Loss = 0.0193   Validation Loss = 0.0184
Epoch  880 of 1000:   Train Loss = 0.0193   Validation Loss = 0.0184
Epoch  920 of 1000:   Train Loss = 0.0193   Validation Loss = 0.0184
Epoch  960 of 1000:   Train Loss = 0.0193   Validation Loss = 0.0184
Epoch 1000 of 1000:   Train Loss = 0.0193   Validation Loss = 0.0183
```


Training
Validation
Loss
3
2
1
0
0
200
400
600
800
1000
Epoch

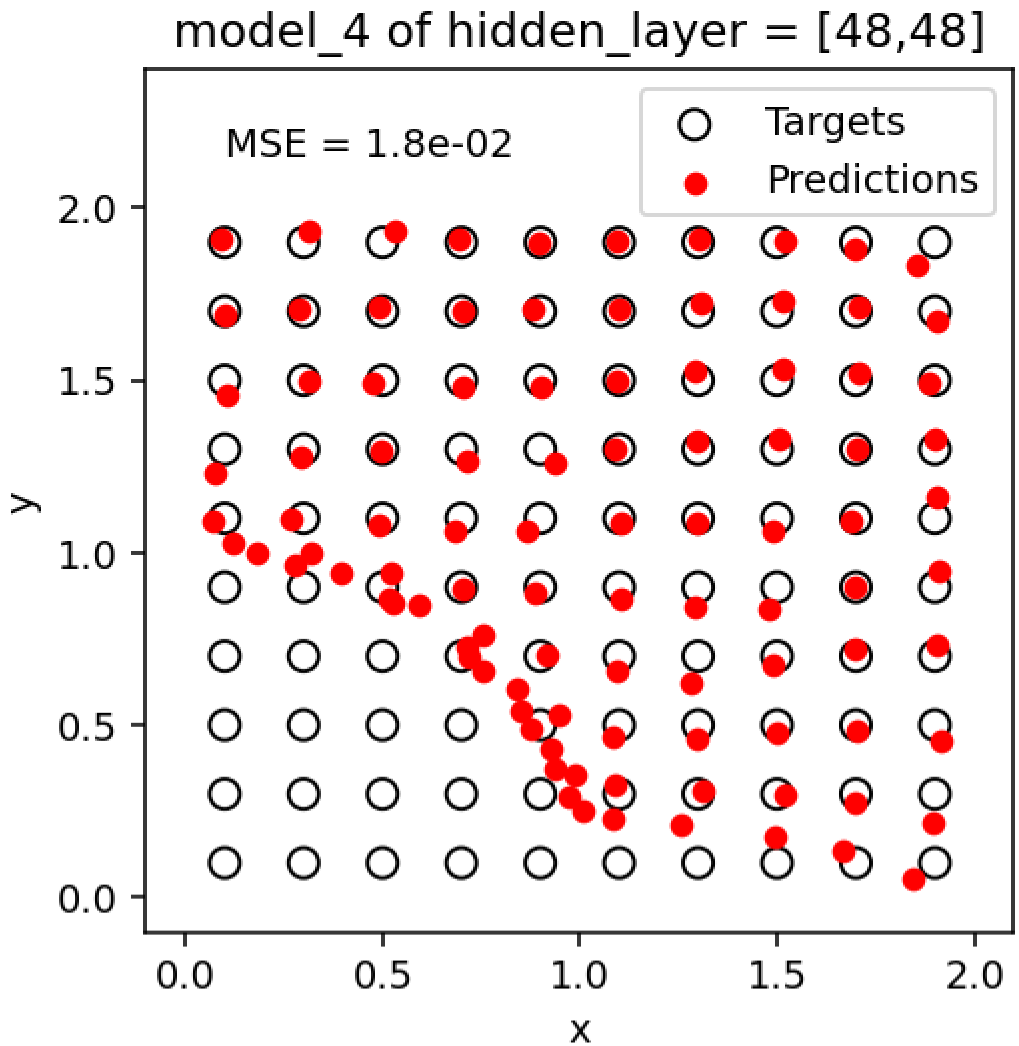
```
Epoch    0 of 1000:   Train Loss = 2.8851   Validation Loss = 2.8757
Epoch   40 of 1000:   Train Loss = 0.6252   Validation Loss = 0.6325
Epoch   80 of 1000:   Train Loss = 0.3433   Validation Loss = 0.3457
Epoch  120 of 1000:   Train Loss = 0.3396   Validation Loss = 0.3423
Epoch  160 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  200 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  240 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  280 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  320 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  360 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  400 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  440 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  480 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  520 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  560 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  600 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  640 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  680 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  720 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  760 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  800 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  840 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  880 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  920 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch  960 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
Epoch 1000 of 1000:   Train Loss = 0.3395   Validation Loss = 0.3423
```

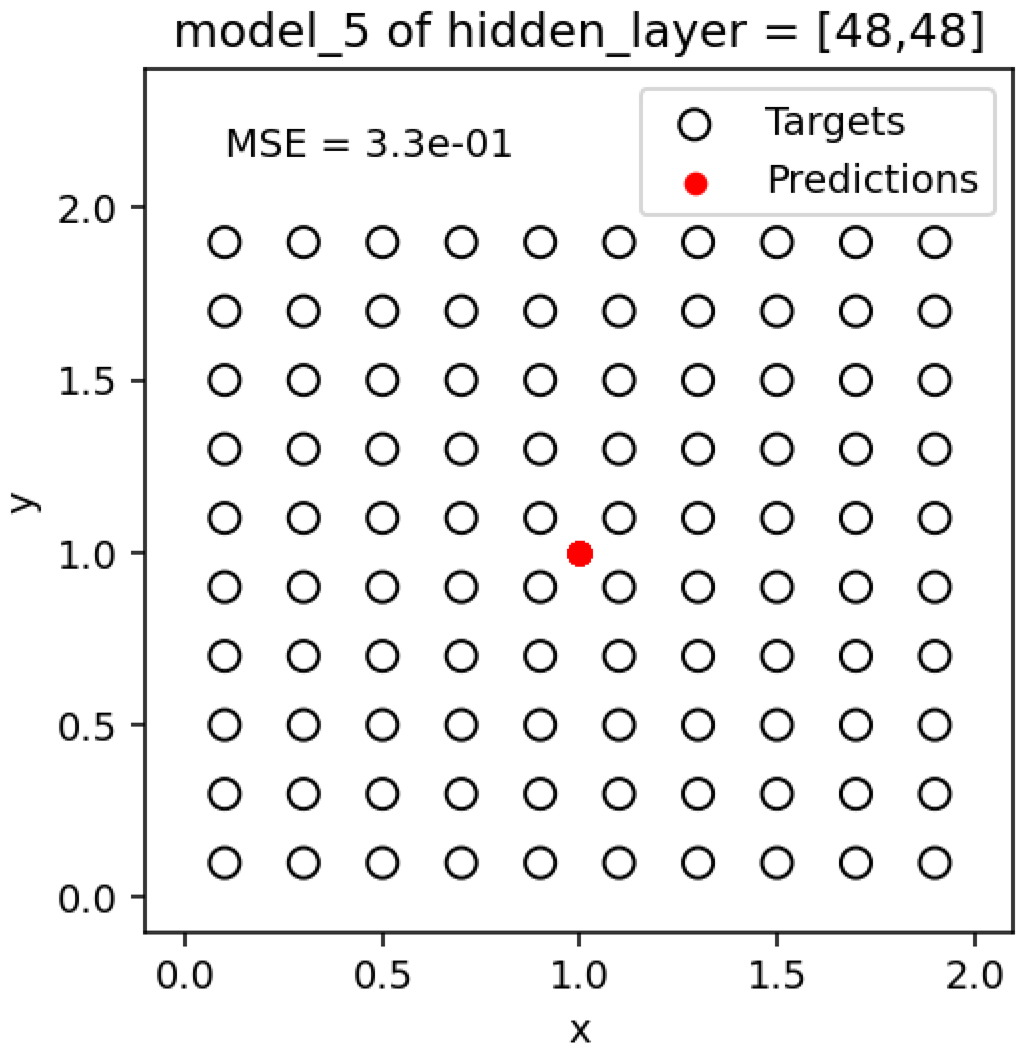

Training
Validation
Loss
4
3
2
1
0
200
400
600
800
1000
Epoch

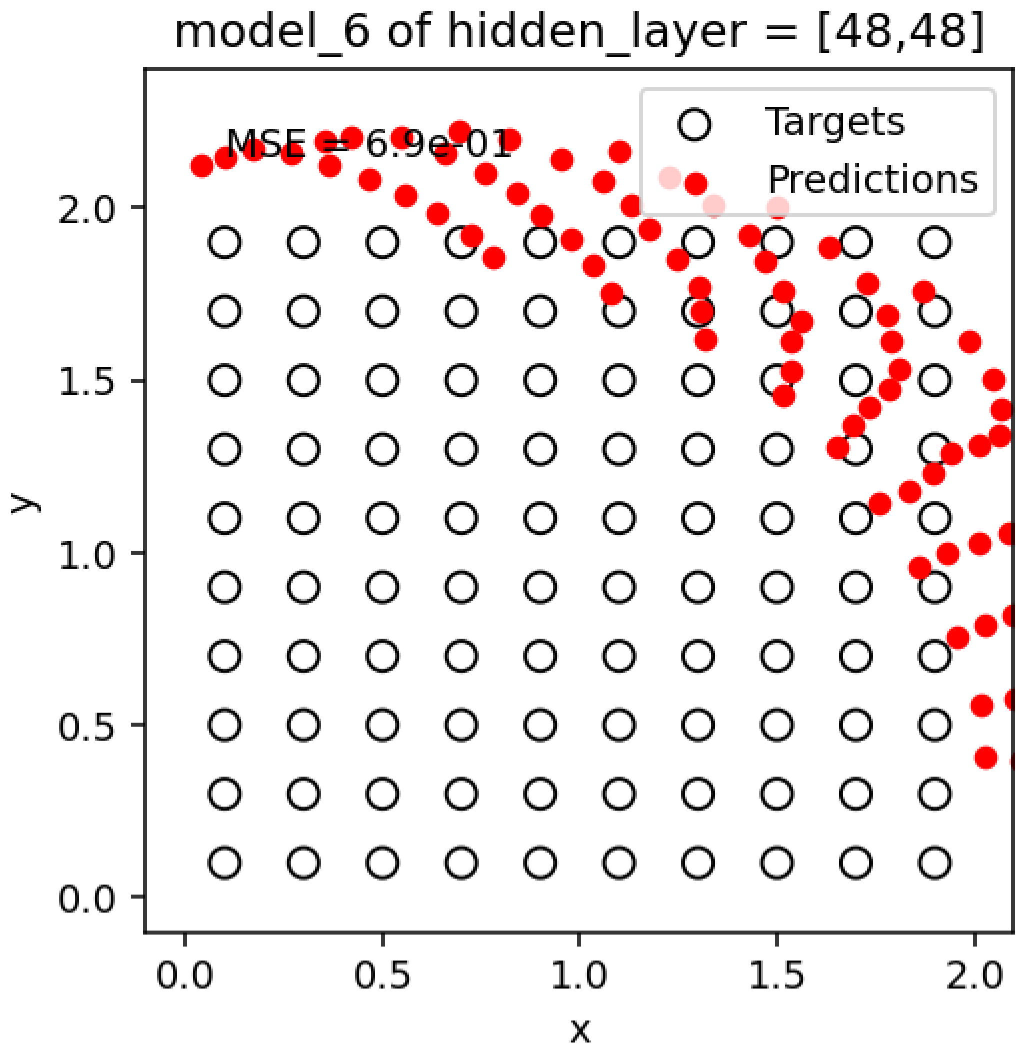
```
Epoch   0 of 300:   Train Loss = 4.7660   Validation Loss = 4.7536
Epoch  12 of 300:   Train Loss = 4.3025   Validation Loss = 4.2910
Epoch  24 of 300:   Train Loss = 3.8410   Validation Loss = 3.8301
Epoch  36 of 300:   Train Loss = 3.3880   Validation Loss = 3.3783
Epoch  48 of 300:   Train Loss = 2.9441   Validation Loss = 2.9354
Epoch  60 of 300:   Train Loss = 2.5200   Validation Loss = 2.5135
Epoch  72 of 300:   Train Loss = 2.1450   Validation Loss = 2.1407
Epoch  84 of 300:   Train Loss = 1.8367   Validation Loss = 1.8346
Epoch  96 of 300:   Train Loss = 1.5995   Validation Loss = 1.5994
Epoch 108 of 300:   Train Loss = 1.4304   Validation Loss = 1.4319
Epoch 120 of 300:   Train Loss = 1.3197   Validation Loss = 1.3224
Epoch 132 of 300:   Train Loss = 1.2484   Validation Loss = 1.2518
Epoch 144 of 300:   Train Loss = 1.1998   Validation Loss = 1.2036
Epoch 156 of 300:   Train Loss = 1.1626   Validation Loss = 1.1665
Epoch 168 of 300:   Train Loss = 1.1298   Validation Loss = 1.1337
Epoch 180 of 300:   Train Loss = 1.0978   Validation Loss = 1.1016
Epoch 192 of 300:   Train Loss = 1.0652   Validation Loss = 1.0689
Epoch 204 of 300:   Train Loss = 1.0312   Validation Loss = 1.0348
Epoch 216 of 300:   Train Loss = 0.9957   Validation Loss = 0.9991
Epoch 228 of 300:   Train Loss = 0.9584   Validation Loss = 0.9616
Epoch 240 of 300:   Train Loss = 0.9193   Validation Loss = 0.9222
Epoch 252 of 300:   Train Loss = 0.8782   Validation Loss = 0.8808
Epoch 264 of 300:   Train Loss = 0.8348   Validation Loss = 0.8372
Epoch 276 of 300:   Train Loss = 0.7892   Validation Loss = 0.7915
Epoch 288 of 300:   Train Loss = 0.7423   Validation Loss = 0.7444
Epoch 300 of 300:   Train Loss = 0.6958   Validation Loss = 0.6978
```




In [10]:
```
plot_predictions(model4,title = "model_4 of hidden_layer = [48,48]")
plot_predictions(model5,title = "model_5 of hidden_layer = [48,48]")
plot_predictions(model6,title = "model_6 of hidden_layer = [48,48]")
```


model_4 of hidden_layer = [48,48]
MSE = 1.8e-02
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x


model_5 of hidden_layer = [48,48]
MSE = 3.3e-01
Targets
Predictions
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x




## Prompts

Neither of these models should have great performance. Describe what went wrong in each case.

Model 1:

The max angle is set at pi/2 which is 90 deg and the tanh(90) is 0.914 around. As the max angle is set to pi/2 so there is no way the prediction value will reach the origin point therefore the clustering happens around the circle i,e., near 1 on both y and x axis.

Model 2:

The learning rate is so high and the epochs really small so the model is not able to train and the loss never converges. As the learning rate is very high, it takes the average of one point and gets stuck and then the MSE remains same for all the epochs. Hence, on the prediction plot, it shows only one point. Therefore, not giving us the correct value.

Model 3:

The prediction plot shows that it is clustering around the top part of the graph. This is because the learning rate is very small and the epochs are not high enough. Along with that as the max angle is set at 2 so there is no way that the prediction value will reach the origin point and the clustering happens around the circle i.e., near 2 on both x and y axis.

In [ ]: